

[Home](#)[Seminars](#)[Free Newsletter](#)[Back Issues](#)[Jack's Books](#)[Jack's Articles](#)[Contact Us](#)

## Smoothing Digital Inputs

There's a lot of way ways to debounce digital inputs. A few are listed in this article.

Published in Embedded Systems Programming, October 1992

- [Return](#) to articles list
- [Return](#) to TGG's home page

For hints, tricks and ideas about better ways to build embedded systems, subscribe to **Embedded Muse**, a free biweekly e- newsletter. No advertising, just down to earth embedded talk.

To subscribe, enter your zip code and email address, and press "Sign Up".

Zip or Country

Email

[Sign Up](#)

*Privacy policy:* Despite repeated requests and even offers of cold hard cash, we never sell or make your email address or other information available to any other party.

The gurus of analog electronics sometimes wax poetic about the beauty of their craft. It's not unusual to see a beatific smile crossing their faces as they tune a feedback loop to tweak the last little bit of stability into an amplifier. Personally, I detest the analog world so full of, well, reality. The analog world is full of noise and oscillation, components degrade as they age, and all the other problems that seem unrelated to making a lousy circuit work.

Once, I thought computers were immune to these problems. Now we're all fiddling with digital phase locked loops that don't lock and high speed circuits that make even the simplest computer an RF nightmare. Even well designed digital circuits are subject to noise effects. Noise makes analog designers old and grey. It can have the same effect on the unwary firmware engineer.

Our realm is the world of ones and zeroes. There are no half levels (at least not for logic). The computer will make a binary decision about any input applied to it. This does not mean, however, that the logic 1 your code just read from some peripheral really means the device is supplying a one.

Some time ago I put a measurement system in a steel mill. Like many big industrial processes, steel making creates a staggering amount of electromagnetic interference. A 1000 foot line consisted of perhaps 400 huge rollers, each weighing several tons, and equipped with a 10 horsepower motor. The motors drove a steel plate back and forth reversing direction every few seconds. The EMF generated during a direction reversal was enough to induce several volts of signal on even a simple unconnected voltmeter. Our sensors brought 20 to 100 volts of induced spikes back to the computer room on their cabling. Despite careful differential design, erratic values from these digital inputs were uncommon.

It's generally pretty safe to assume that digital signals generated locally to the embedded controller are clean and free from noise-induced transients. Any input, whether digital or analog, that comes over a long cable should be looked at suspiciously. If the signal changes to a random value occasionally, will it wreak havoc in your product?

### Noise

What exactly is noise? Analogers consider any unwanted signal alteration a form of noise, though generally noise is defined as that part of the input originating from something other than the sensor. For example, cosmic background radiation was discovered by an engineer working for the phone company, who found interference no matter which way they oriented their antennas. Very recently the Cosmic Background Explorer satellite mapped this radiation and found compelling evidence supporting the big bang theory of cosmology.

Digital noise is the same thing. Given a 1 from the sensor, the signal is noise-free if the computer always reads that value as a one. If once in a hundred readings the firmware interprets the signal as a zero, then the channel is not noiseless, so either the code or the electronics will need some tuning to correct the problem.

Mechanical contacts used as computer inputs are a prime source of "noise", though this noise is referred to as "bounce". A switch or relay contact literally chatters for quite a few milliseconds when it closes. That is, pressing a switch will send a random stream of ones and zeroes to the computer for a long time.

Sometimes pure digital signals are transmitted through slip rings. For example, a radar antenna rotates continuously in one direction - it's impossible to wire directly to the antenna, as a cable would quickly twist itself all around the motor and supporting structure. Slip rings are brushes that rub on an insulated shaft to transmit data or power through the rubbing contact. Slip rings, especially when dirty, generate all sorts of interesting modifications to the signals propagated through them.

Another form of "noise" is signal corruption due to limits in the accuracy or dependability of the input source. A compact disk supplies a stream of ones and zeroes, but the reliability of the data is moderately low - pits and fissures in the medium masquerade as real data. This problem is so severe that significant extra hardware and recording space is devoted to error correction codes that can both detect and then correct the flaw. We see the same sort of effect with dynamic RAMs - every PC in the world uses ECC (even parity) on each byte of storage, encoding parity information in one position. Again, the problem is severe (though memory does seem super reliable now) that the cost of a lot of extra hardware is justified.

It is pretty common to see error detection bits appended to nearly all serial communication schemes. Few simple parallel input bits get the same sort of treatment. Is this because application can tolerate errors? Or, have the designers merely forgotten to question the basic assumptions? Remember: nature is perverse, and will endeavor to cause maximum pain at the worst time. If there is any chance that your encoder or switch inputs could become garbled occasionally, then be awfully sure your code can deal with the problem.

#### **Hardware Solutions**

To my knowledge, there are only three ways a hardware designer can deal with noisy digital inputs. The first is to use the best possible practices for transmitting signal. This includes routing signal wires away from noise-inducing high current cables, adding shielding where necessary, and using the proper transmission techniques. A fiber optic link is immune from induced EMF and is an ideal solution.

A lot of folks assume differential transmitters and receivers provide foolproof transmission in any environment. The theory is that both the positive and negative signals (the data is sent on a pair of wires, with a positive and inverted version on each wire) will have the same signal induced. This is called "common mode", since a common corruption appears on each wire. The receiver compares the wires and removes this common mode, hopefully giving a nice clean version of the input. Differential transmission does work very well most of the time, but enough common mode can swamp receivers, and has even been known to destroy the ICs.

Alternatively, good design practice might dictate replacing noisy sensors with something producing a cleaner output, albeit at a higher price. A Hall-Effect switch, for example, produces bounceless outputs. I'm not advocating giving up switches with mechanical contacts, but in some situations it pays to consider alternatives.

A second hardware solution is to transmit either a correction code, as in the case of a parity bit or simply a parity bit or hamming code that at least alerts the computer that the byte simply cannot be correct. Both are great ideas; both are relatively expensive in printed circuit board real estate, copper costs, and circuits used. It's pretty rare to see simple parallel inputs accompanied by ECC or parity information.

A third approach adds hardware to the receiving computer to clean up the input. About the only time you'll see this in switch debouncing applications, where a simple set-reset flop removes all switch bounce. The downside is the use of half a 7400 for each switch and the need for double pole switches.

While on the subject of hardware, I'll get on my anti-NMI soapbox again and ask everyone to NOT connect NMI, or any other interrupt line for that matter, to a mechanical contact. The only exception is if you've added a hardware debouncer to the signal. Otherwise every bit of bounce will reinitiate the service routine. With NMI one real interrupt will masquerade as several independent interrupts, each one pushing on the stack and recalling the ISR. Maskable interrupts will be serviced on every bounce, needlessly eating up valuable CPU time.

#### **Firmware Fixes**

A lot of systems remove analog noise by averaging the input; this is not really an option



for smoothing digital inputs, because a switch can only be on or off; it cannot (except fuzzy logic systems, I suppose) be 50% on. Thus, digital filters always seem to be very simple algorithms: if 75 of 100 reads of the input sense a logic one, then the code assumes that the data is indeed a one.

Other algorithms read and reread until the data settles down. If 75 of 100 reads give a logic one, then the input is noisy and is resampled until 100 of 100 reads are consistent.

The problem with the trivial implementations is that each requires a lot of elapsed time. Take the case of debouncing a switch. The switch is going to bounce practically forever, perhaps as long as 50-100 msec. Nothing the code can do will eliminate this delay. Software that reads and reads and reads until 100 of 100 reads are identical will burn time until that inevitable 50-100 msec elapses.

One way to eliminate this wasteful use of the CPU is to have a timer interrupt the CPU regularly, read and filter the input(s), and store the result for future use.

I've always felt the one hardware resource I want to support the software is a timer. Programmed to interrupt every 1 to 10 msec, it provides a time base that is very useful for sequencing tasks and other activities. Digital filtering is a perfect use of the timer, as in the case of debouncing) a few reads widely separated in time are every bit as good as a thousand sequential reads.

Even better than writing a big, complex interrupt service routine that filters the input is to use a Real Time Operating System (RTOS). I'll ruefully admit to having rolled my own RTOS on more than one occasion. Usually, my homebrewn ones are fairly primitive, supporting multitasking but little else. I buy DOS, I buy word processors; why write my own RTOS?

Tyler Sperry and Gretchen Bay reported on RTOSs in an earlier issue. If you have not looked at what a commercially available RTOS brings to the party, by all means get some vendor literature and look at the rich resources they provide. Multitasking is only a small part of the functionality of an RTOS. I'm particularly fascinated by the message passing capabilities of these products.

The OOP movement advocates building brick walls around functions. Keep variables local to functions, and pass parameters around as messages. It's a religious experience to read code written in any language using the messaging philosophy. Global variables, the bane of a lot of problem code, will be non-existent. Each function gets everything it needs from a parameter list.

A decent RTOS supports all sorts of message passing, including queues, semaphores, and "messages" - data the RTOS passes between tasks. The RTOS takes care of all of the details of message traffic; your code merely has to issue an RTOS call to send one, and issue another call to receive the message. Global variables for intertask communication are eliminated, along with the potential for a rogue routine that slyly alters the global and screws everything up.

An RTOS bundled into an embedded system will make debouncing and filtering much cleaner and faster. Use the timer ticks to drive the RTOS and initiate context switching.

Then, write the filter as a task that generates a smoothed version of the input into a variable local to the task. If the main line code needs the current smoothed input, it can request the task to reply with the data via a message. This keeps the entire filtering code and all of its variables in one well-insulated module. Once debugged, you'll never have to look at it again. No other task should be able to effect the operation of the filter.

As an aside, some systems read voltage controlled oscillators (VCOs) and compute the frequency of the signal by counting edges over some period of time. An RTOS is the perfect vehicle for this as well - the rationale and methods are practically the same as digital filtering. One task can count edges; another, running at some fixed rate, can read the count and scale by the time base. Again, if the result is stored into a static local variable, any other activity in the program can request the current frequency without being forced to wait for the data to be acquired.

If you do feel compelled to write non-interrupt based filters, be wary of the delay code. Typically these algorithms (especially debouncing code) do a number of reads, wait a while, and read some more to see if the input settles down. This makes sense but has coding delays that are always perilous.

Is the delay long enough? How do you know for sure? If that short loop takes only 1 microsecond, not the millisecond you assumed, will this cause some erroneous read? In assembly language it's easy to figure the number of T-states in a loop to get a good handle on the time involved, but a small code change several years hence may disrupt these figures. Or, changing CPUs, clock rates, or even the number of wait states on a new version of the product will cause similar problems.

Writing in C is more of a problem. How long does 100 repetitions of a null FOR loop anyway? This will change depending on the optimizations, compiler version, and even memory model selected. At the very least use a scope to measure the real delay. Preferably, generate delays via an interrupt-driven timer.

#### Conclusion

If a switch bounces as it is closed, can we shorten the time to decide if it is on or off by looking at the relative frequencies of ones and zeroes? It seems to me that as the switch starts to settle down we'd see many more ones than zeroes returned. Has anyone experienced this? Feel free to contact me.

[Back](#) to home page.

#### *The Ganssle Group*

PO Box 38346, Baltimore, MD 21231

Tel: 410-496-3647, Fax: 647-439-1454

Email [info@ganssle.com](mailto:info@ganssle.com)

© 2000, 2001, 2002, 2003, 2004, 2005 The Ganssle Group